7 Model binding and validation in minimal APIs

This chapter covers

- Using request values to create binding models
- Customizing the model-binding process
- Validating user input using DataAnnotations attributes

In chapter 6 I showed you how to define a route with parameters—perhaps for the unique ID for a product API. But say a client sends a request to the product API. What then? How do you access the values provided in the request and read the JavaScript Object Notation (JSON) in the request body?

For most of this chapter, in sections 7.1-7.9, we'll look at model binding and how it simplifies reading data from a request in minimal APIs. You'll see how to take the data posted in the request body or in the URL and bind it to C# objects, which are then passed to your endpoint handler methods as arguments. When your handler executes, it can use these values to do something useful—return a product's details or change a product's name, for example.

When your code is executing in an endpoint handler method, you might be forgiven for thinking that you can happily use the binding model without any further thought. Hold on, though. Where did that data come from? From a user—and you know users can't be trusted! Section 7.10 focuses on how to make sure that the user-provided values are valid and make sense for your app.

Model binding is the process of taking the user's raw HTTP request and making it available to your code by populating plain old CLR objects (POCOs), providing the input to your endpoint handlers. We start by looking at which values in the request are available for binding and where model binding fits in your running app.

7.1 Extracting values from a request with model binding

In chapters 5 and 6 you learned that route parameters can be extracted from the request's path and used to execute minimal API handlers. In this section we look in more detail at the process of extracting route parameters and the concept of model binding.

By now, you should be familiar with how ASP.NET Core handles a request by executing an endpoint handler. You've also already seen several handlers, similar to

```
app.MapPost("/square/{num}", (int num) => num * num);
```

Endpoint handlers are normal C# methods, so the ASP.NET Core framework needs to be able to call them in the usual way. When handlers accept parameters as part of their method signature, such as `num` in the preceding example, the framework needs a way to generate those objects. Where do they come from, exactly, and how are they created?

I've already hinted that in most cases, these values come from the request itself. But the HTTP request that the server receives is a series of strings. How does ASP.NET Core turn that into a .NET object? This is where model binding comes in.

> **DEFINITION** *Model binding* extracts values from a request and uses them to create .NET objects. These objects are passed as method parameters to the endpoint handler being executed.

The model binder is responsible for looking through the request that comes in and finding values to use. Then it creates objects of the appropriate type and assigns these values to your model in a process called *binding*.

> **NOTE** Model binding in minimal APIs (and in Razor Pages and Model-View-Controller [MVC]) is a one-way population of objects from the request, not the two-way data binding that desktop or mobile development sometimes uses.

ASP.NET Core automatically creates the arguments that are passed to your handler by using the request's properties, such as the request URL, any headers sent in the HTTP request, any data explicitly `POST`ed in the request body, and so on.

Model binding happens before the filter pipeline and your endpoint handler execute, in the `EndpointMiddleware`, as shown in figure 7.1. The `RoutingMiddleware` is

responsible for matching an incoming request to an endpoint and for extracting the route parameter values, but all the values at that point are `string`s. It's only in the `EndpointMiddleware` that the `string` values are converted to the real argument types (such as `int`) needed to execute the endpoint handler.

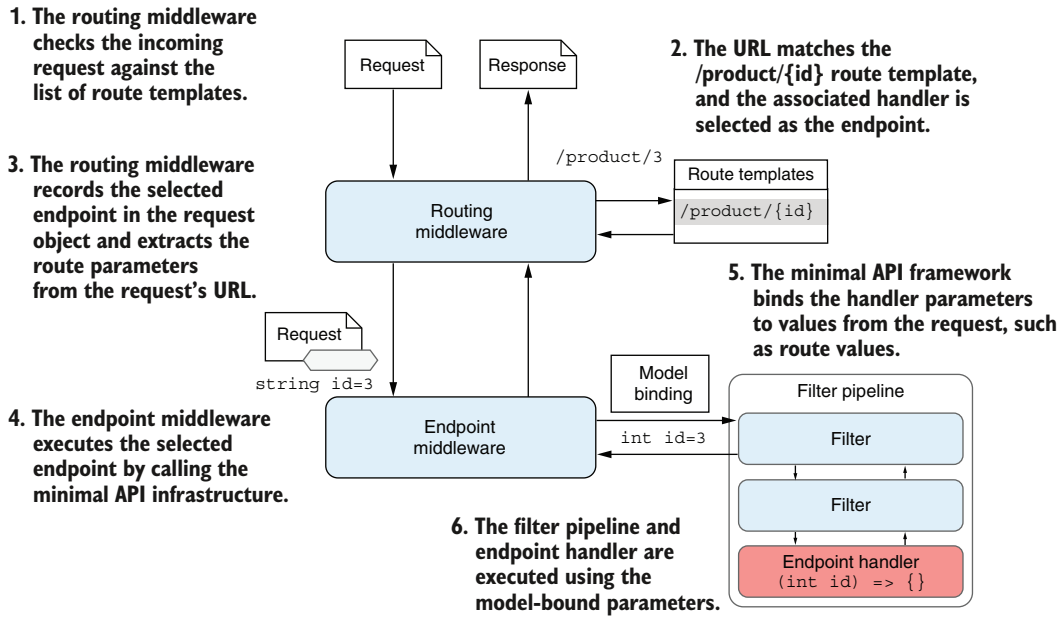


Figure 7.1 The `RoutingMiddleware` matches the incoming request to an endpoint and extracts the route parameters as `string`s. When the `EndpointMiddleware` executes the endpoint, the minimal API infrastructure uses model binding to create the arguments required to execute the endpoint handler, converting the `string` route values to real argument types such as `int`.

For every parameter in your minimal API endpoint handler, ASP.NET core must decide how to create the corresponding arguments. Minimal APIs can use six different binding sources to create the handler arguments:

- *Route values*—These values are obtained from URL segments or through default values after matching a route, as you saw in chapter 5.
- *Query string values*—These values are passed at the end of the URL, not used during routing.
- *Header values*—Header values are provided in the HTTP request.
- *Body JSON*—A single parameter may be bound to the JSON body of a request.
- *Dependency injected services*—Services available through dependency injection can be used as endpoint handler arguments. We look at dependency injection in chapters 8 and 9.
- *Custom binding*—ASP.NET Core exposes methods for you to customize how a type is bound by providing access to the `HttpRequest` object.

WARNING Unlike MVC controllers and Razor Pages, minimal APIs do *not* automatically bind to the body of requests sent as forms, using the `application/x-www-form-urlencoded` mime type. Minimal APIs will bind only to a JSON request body. If you need to work with form data in a minimal API endpoint, you can access it on `HttpRequest.Form`, but you won't benefit from automatic binding.

We'll look at the exact algorithm ASP.NET Core uses to choose which binding source to use in section 7.8, but we'll start by looking at how ASP.NET Core binds simple types such as `int` and `double`.

7.2 Binding simple types to a request

When you're building minimal API handlers, you'll often want to extract a simple value from the request. If you're loading a list of products in a category, for example, you'll likely need the category's ID, and in the calculator example at the start of section 7.1, you'll need the number to square.

When you create an endpoint handler that contains simple types such as `int`, `string`, and `double`, ASP.NET Core automatically tries to bind the value to a route parameter, or a query string value:

- If the name of the handler parameter matches the name of a route parameter in the route template, ASP.NET Core binds to the associated route value.
- If the name of the handler parameter doesn't match any parameters in the route template, ASP.NET Core tries to bind to a query string value.

If you make a request to `/products/123`, for example, this will match the following endpoint:

```
app.MapGet("/products/{id}", (int id) => $"Received {id}");
```

ASP.NET Core binds the `id` handler argument to the `{id}` route parameter, so the handler function is called with `id=123`. Conversely, if you make a request to `/products?id=456`, this will match the following endpoint instead:

```
app.MapGet("/products", (int id) => $"Received {id}");
```

In this case, there's no `id` parameter in the route template, so ASP.NET Core binds to the query string instead, and the handler function is called with `id=456`.

In addition to this "automatic" inference, you can force ASP.NET Core to bind from a specific source by adding attributes to the parameters. `[FromRoute]` explicitly binds to route parameters, `[FromQuery]` to the query string, and `[FromHeader]` to header values, as shown in figure 7.2.

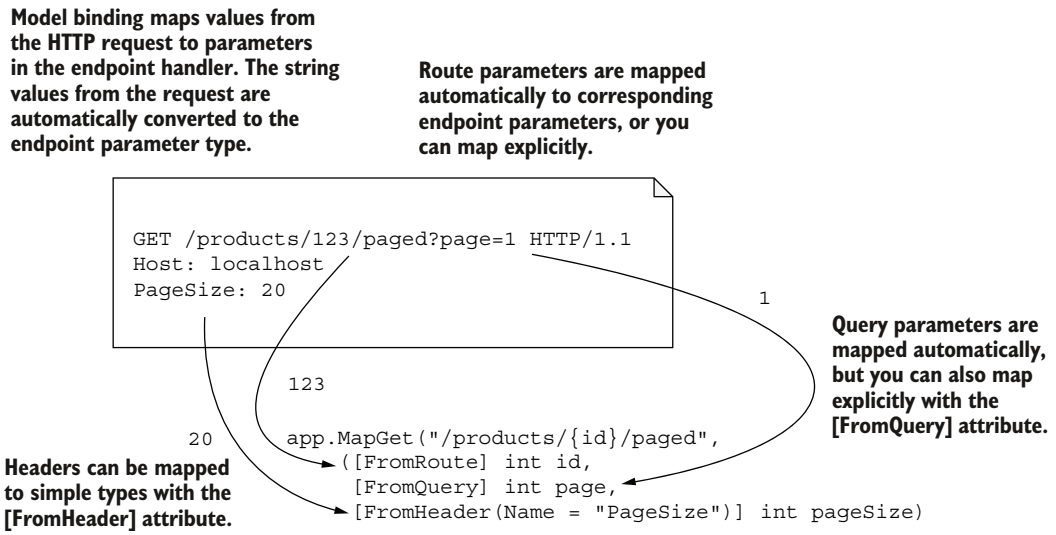


Figure 7.2 Model binding an HTTP get request to an endpoint. The `[FromRoute]`, `[FromQuery]`, and `[FromHeader]` attributes force the endpoint parameters to bind to specific parts of the request. Only the `[FromHeader]` attribute is required in this case; the route parameter and query string would be inferred automatically.

The `[From*]` attributes override ASP.NET Core's default logic and forces the parameters to load from a specific binding source. Listing 7.1 demonstrates three possible `[From*]` attributes:

- `[FromQuery]`—As you've already seen, this attribute forces a parameter to bind to the query string.
- `[FromRoute]`—This attribute forces the parameter to bind a route parameter value. Note that if a parameter of the required name doesn't exist in the route template, you'll get an exception at runtime.
- `[FromHeader]`—This attribute binds a parameter to a header value in the request.

Listing 7.1 Binding simple values using `[From]` attributes

```
using Microsoft.AspNetCore.Mvc;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/products/{id}/paged",
    ([FromRoute] int id,
     [FromQuery] int page,
     [FromHeader(Name = "PageSize")] int pageSize)
    => $"Received id {id}, page {page}, pageSize {pageSize}");

app.Run();
```

All the [From*] attributes are in this namespace.

[FromRoute] forces the argument to bind to the route value.

[FromQuery] forces the argument to bind to the query string.

[FromHeader] binds the argument to the specified header.

Later, you'll see other attributes, such as `[FromBody]` and `[FromServices]`, but the preceding three attributes are the only `[From*]` attributes that operate on simple types such as `int` and `double`. I prefer to avoid using `[FromQuery]` and `[FromRoute]` wherever possible and rely on the default binding conventions instead, as I find that they clutter the method signatures, and it's generally obvious whether a simple type is going to bind to the query string or a route value.

> **TIP** ASP.NET Core binds to route parameters and query string values based on convention, but the only way to bind to a header value is with the `[FromHeader]` attribute.

You may be wondering what would happen if you try to bind a type to an incompatible value. What if you try to bind an `int` to the string value `"two"`, for example? In that case ASP.NET Core throws a `BadHttpRequestException` and returns a `400 Bad Request` response.

> **NOTE** When the minimal API infrastructure fails to bind a handler parameter due to an incompatible format, it throws a `BadHttpRequestException` and returns a `400 Bad Request` response.

I've mentioned several times in this section that you can bind route values, query string values, and headers to simple types, but what *is* a simple type? A *simple type* is defined as any type that contains either of the following `TryParse` methods, where `T` is the implementing type:

```
public static bool TryParse(string value, out T result);
public static bool TryParse(
    string value, IFormatProvider provider, out T result);
```

Types such as `int` and `bool` contain one (or both) these methods. But it's also worth noting that you can create your own types that implement one of these methods, and they'll be treated as simple types, capable of binding from route values, query string values, and headers.

Figure 7.3 shows an example of implementing a simple strongly-typed ID[1] that's treated as a simple type thanks to the `TryParse` method it exposes. When you send a request to `/product/p123`, ASP.NET Core sees that the `ProductId` type used in the endpoint handler contains a `TryParse` method and that the name of the `id` parameter has a matching route parameter name. It creates the `id` argument by calling `ProductId.TryParse()` and passes in the route value, `p123`.

Listing 7.2 shows how you could implement the `TryParse` method for `ProductId`. This method creates a `ProductId` from `string`s that consist of an `integer` prefixed with `'p'` (`p123` or `p456`, for example). If the input `string` matches the required format, it

[1] I have a series discussing strongly-typed IDs and their benefits on my blog at http://mng.bz/a1Kz.

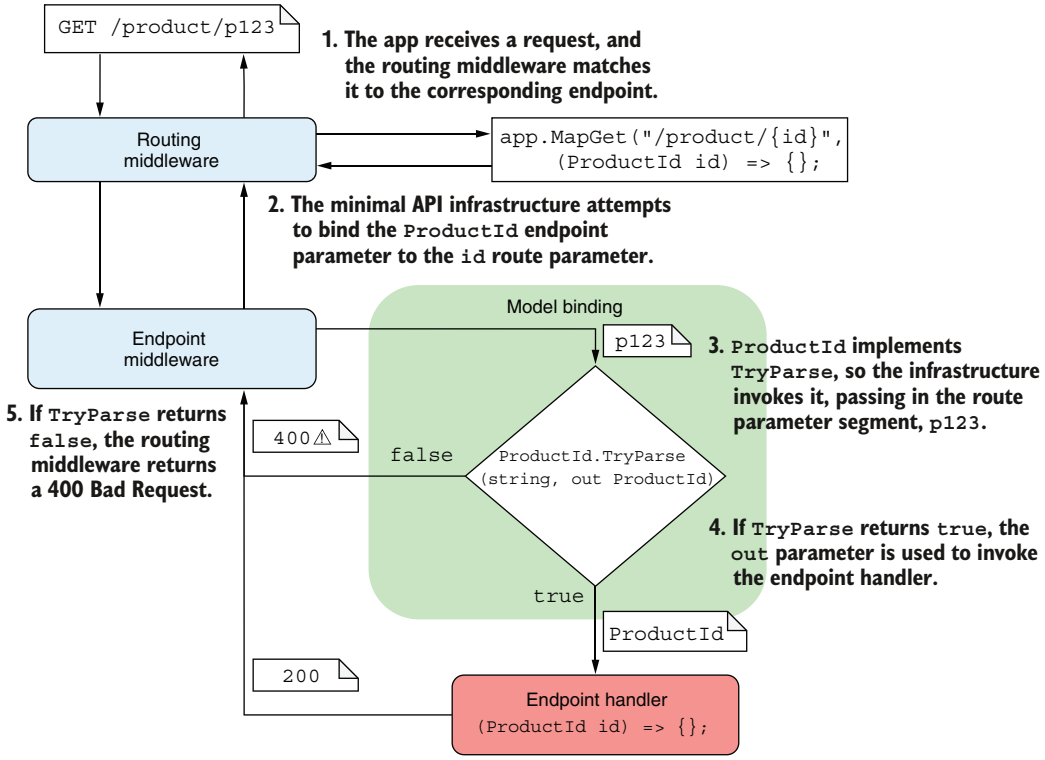


Figure 7.3 The routing middleware matches the incoming URL to the endpoint. The endpoint middleware attempts to bind the route parameter `id` to the endpoint parameter. The endpoint parameter type `ProductId` implements `TryParse`. If parsing is successful, the parsed parameter is used to call the endpoint handler. If parsing fails, the endpoint middleware returns a `400 Bad Request` response.

creates a `ProductId` instance and returns `true`. If the format is invalid, it returns `false`, binding fails, and a `400 Bad Request` is returned.

Listing 7.2 Implementing `TryParse` in a custom type to allow parsing from route values

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/product/{id}", (ProductId id) => $"Received {id}");
app.Run();

readonly record struct ProductId(int Id)
{
    public static bool TryParse(string? s, out ProductId result)
    {
        if(s is not null
            && s.StartsWith('p')
            && int.TryParse(
```

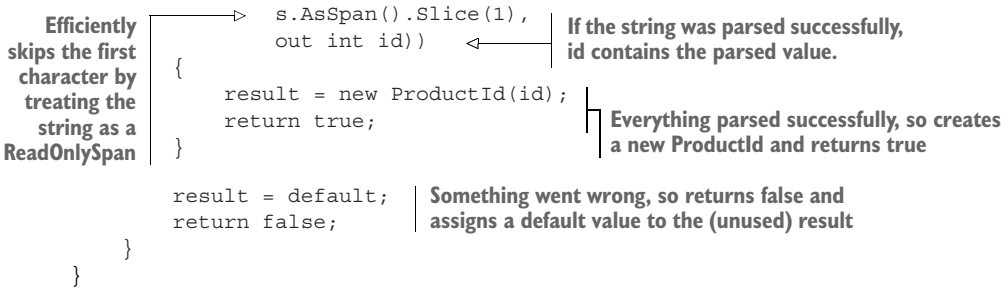
ProductId automatically binds to route values as it implements TryParse.

ProductId is a C# 10 record struct.

It implements TryParse, so it's treated as a simple type by minimal APIs.

Checks that the string is not null and that the first character in the string is 'p' . . .

and if it is, tries to parse the remaining characters as an integer



Using modern C# and .NET features

Listing 7.2 included some C# and .NET features that you may not have seen before, depending on your background:

- *Pattern matching for null values*—`s is not null`. Pattern matching features have been introduced gradually into C# since C# 7. The `is not null` pattern, introduced in C# 9, has some minor advantages over the common `!= null` expression. You can read all about pattern matching at http://mng.bz/gBxl.
- *Records and struct records*—`readonly record struct`. Records are syntactical sugar over normal `class` and `struct` declarations, which make declaring new types more succinct and provide convenience methods for working with immutable types. Record structs were introduced in C# 10. You can read more at http://mng.bz/5wWz.
- `Span<T>` *for performance*—`s.AsSpan()`. `Span<T>` and `ReadOnlySpan<T>` were introduced in .NET Core 2.1 and are particularly useful for reducing allocations when working with `string` values. You can read more about them at http://mng.bz/6DNy.
- `ValueTask<T>`—It's not shown in listing 7.2, but many of the APIs in ASP.NET Core use `ValueTask` instead of the more common `Task` for APIs that normally complete asynchronously but *may* complete asynchronously. You can read about why they were introduced and when to use them at http://mng.bz/o1GM.

Don't worry if you're not familiar with these constructs. C# is a fast-moving language, so keeping up can be tricky, but there's generally no reason you need to use the new features. Nevertheless, it's useful to be able to recognize them so that you can read and understand code that uses them.

If you're keen to embrace new features, you might consider implementing the `IParsable` interface when you implement `TryParse`. This interface uses the `static abstract interfaces` feature, which was introduced in C# 11, and requires implementing both a `TryParse` and `Parse` method. You can read more about the `IParsable` interface in the announcement post at http://mng.bz/nW2K.

Now we've looked extensively at binding simple types to route values, query strings, and headers. In section 7.3 we'll learn about binding to the body of a request by deserializing JSON to complex types.

7.3 Binding complex types to the JSON body

Model binding in minimal APIs relies on certain conventions to simplify the code you need to write. One such convention, which you've already seen, is about binding to route parameters and query string values. Another important convention is that minimal API endpoints assume that requests will be sent using JSON.

Minimal APIs can bind the body of a request to a single complex type in your endpoint handler by deserializing the request from JSON. That means that if you have an endpoint such as the one in the following listing, ASP.NET Core will automatically deserialize the request for you from JSON, creating the `Product` argument.

Listing 7.3 Automatically deserializing a JSON request from the body

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/product", (Product product) => $"Received {product}");

app.Run();

record Product(int Id, string Name, int Stock);
```

Product is a complex type, so it's bound to the JSON body of the request.

Product doesn't implement TryParse, so it's a complex type.

If you send a `POST` request to `/product` for the app in listing 7.3, you need to provide valid JSON in the request body, such as

```
{ "id": 1, "Name": "Shoes", "Stock": 12 }
```

ASP.NET Core uses the built-in System.Text.Json library to deserialize the JSON into a `Product` instance and uses it as the `product` argument in the handler.

Configuring JSON binding with System.Text.Json

The System.Text.Json library, introduced in .NET Core 3.0, provides a high-performance, low-allocation JSON serialization library. It was designed to be something of a successor to the ubiquitous Newtonsoft.Json library, but it trades flexibility for performance.

Minimal APIs use System.Text.Json for both JSON deserialization (when binding to a request's body) and serialization (when writing results, as you saw in chapter 6). Unlike for MVC and Razor Pages, you can't replace the JSON serialization library used by minimal APIs, so there's no way to use Newtonsoft.Json instead. But you can customize some of the library's serialization behavior for your minimal APIs.

You can set System.Text.Json, for example, to relax some of its strictness to allow trailing commas in the JSON and control how property names are serialized with code like the following example:

(continued)

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.ConfigureRouteHandlerJsonOptions(o => {
    o.SerializerOptions.AllowTrailingCommas = true;
    o.SerializerOptions.PropertyNamingPolicy =
            JsonNamingPolicy.CamelCase;
    o.SerializerOptions.PropertyNameCaseInsensitive = true;
});
```

Typically, the automatic binding for JSON requests is convenient, as most APIs these days are built around JSON requests and responses. The built-in binding uses the most performant approach and eliminates a lot of boilerplate that you'd otherwise need to write yourself. Nevertheless, bear several things in mind when you're binding to the request body:

- You can bind only a single handler parameter to the JSON body. If more than one complex parameter is eligible to bind to the body, you'll get an exception at runtime when the app receives its first request.
- If the request body isn't JSON, the endpoint handler won't run, and the `EndpointMiddleware` will return a `415 Unsupported Media Type` response.
- If you try to bind to the body for an HTTP verb that usually doesn't send a body (`GET`, `HEAD`, `OPTIONS`, `DELETE`, `TRACE`, and `CONNECT`), you'll get an exception at runtime. If you change the endpoint in listing 7.3 to `MapGet` instead of `MapPost`, for example, you'll get an exception on your first request, as shown in figure 7.4.
- If you're sure that you want to bind the body of these requests, you can override the preceding behavior by applying the `[FromBody]` attribute to the handler parameter. I strongly advise against this approach, though: sending a body with `GET` requests is unusual, could confuse the consumers of your API, and is discouraged in the HTTP specification (https://www.rfc-editor.org/rfc/rfc9110#name-get).
- It's uncommon to see, but you can also apply `[FromBody]` to a simple type parameter to force it to bind to the request body instead of to the route/query string. As for complex types, the body is deserialized from JSON into your parameter.

We've discussed binding of both simple types and complex types. Unfortunately, now it's time to admit to a gray area: arrays, which can be simple types *or* complex types.

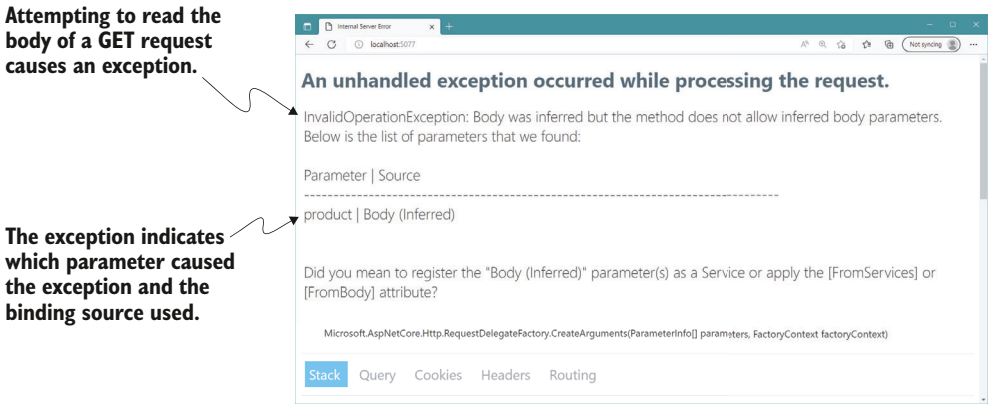


Figure 7.4 If you try to bind the body to a parameter for a `GET` request, you'll get an exception when your app receives its first request.

7.4 Arrays: Simple types or complex types?

It's a little-known fact that entries in the query string of a URL don't have to be unique. The following URL is valid, for example, even though it includes a duplicate `id` parameter:

```
/products?id=123&id=456
```

So how do you access these query string values with minimal APIs? If you create an endpoint like

```
app.MapGet("/products", (int id) => $"Received {id}");
```

a request to `/products?id=123` would bind the `id` parameter to the query string, as you'd expect. But a request that includes two `id` values in the query string, such as `/products?id=123&id=456`, will cause a runtime error, as shown in figure 7.5. ASP.NET Core returns a `400 Bad Request response` without the handler or filter pipeline running at all.

If you want to handle query strings like this one, so that users can optionally pass multiple possible values for a parameter, you need to use arrays. The following listing shows an example of an endpoint that accepts multiple `id` values from the query string and binds them to an array.

Listing 7.4 Binding multiple values for a parameter in a query string to an array

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/products/search",
    (int[] id) => $"Received {id.Length} ids");

app.Run();
```

The array will bind to multiple instances of id in the query string.

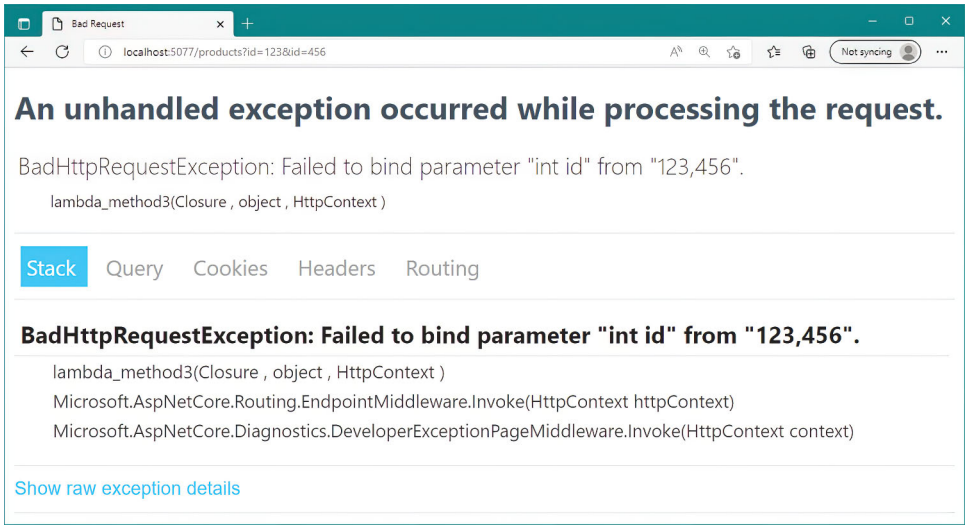


Figure 7.5 Attempting to bind a handler with a signature such as `(int id)` to a query string that contains `?id=123&id=456` causes an exception at runtime and a `400 Bad Request` response.

If you're anything like me, the fact that the `int[]` handler parameter in listing 7.4 is called `id` and not `ids` will really bug you. Unfortunately, you have to use `id` here so that the parameter binds correctly to a query string like `?id=123&id=456`. If you renamed it `ids`, the query string would need to be `?ids=123&ids=456`.

Luckily, you have another option. You can control the name of the target that a handler parameter binds to by using the `[FromQuery]` and `[FromRoute]` attributes, similar to the way you use `[FromHeader]`. For this example, you can have the best of both worlds by renaming the handler parameter `ids` and adding the `[FromQuery]` attribute:

```
app.MapGet("/products/search",
    ([FromQuery(Name = "id")] int[] ids) => $"Received {ids.Length} ids");
```

Now you can sleep easy. The handler parameter has a better name, but it still binds to the query string `?id=123&id=456` correctly.

> **TIP** You can bind array parameters to multiple header values in the same way that you do for as query string values, using the `[FromHeader]` attribute.

The example in listing 7.4 binds an `int[]`, but you can bind an array of any simple type, including custom types with a `TryParse` method (listing 7.2), as well as `string[]` and `StringValues`.

> **NOTE** `StringValues` is a helper type in the `Microsoft.Extensions.Primitives` namespace that represents zero, one, or many `string`s in an efficient way.

So where is that gray area I mentioned? Well, arrays work as I've described only if

- You're using an HTTP verb that typically doesn't include a request body, such as `GET`, `HEAD`, or `DELETE`.
- The array is an array of simple types (or `string[]` or `StringValues`).

If either of these statements is *not* true, ASP.NET Core will attempt to bind the array to the JSON body of the request instead. For `POST` requests (or other verbs that typically have a request body), this process works without problems: the JSON body is deserialized to the parameter array. For `GET` requests (and other verbs without a body), it causes the same unhandled exception you saw in figure 7.4 when a body binding is detected in one of these verbs.

> **NOTE** As before, when binding body parameters, you can work around this situation for `GET` requests by adding an explicit `[FromBody]` to the handler parameter, but you shouldn't!

We've covered binding both simple types and complex types, from the URL and the body, and we've even looked at some cases in which a mismatch between what you expect and what you receive causes errors. But what if a value you expect isn't there? In section 7.5 we look at how you can choose what happens.

7.5 Making parameters optional with nullables

We've described lots of ways to bind parameters to minimal API endpoints. If you've been experimenting with the code samples and sending requests, you may have noticed that if the endpoint *can't* bind a parameter at runtime, you get an error and a `400 Bad Request` response. If you have an endpoint that binds a parameter to the query string, such as

```
app.MapGet("/products", (int id) => $"Received {id}");
```

but you send a request without a query string or with the wrong name in the query string, such as a request to `/products?p=3`, the `EndpointMiddleware` throws an exception, as shown in figure 7.6. The `id` parameter is required, so if it can't bind, you'll get an error message and a `400 Bad Request` response, and the endpoint handler won't run.

All parameters are required regardless of which binding source they use, whether that's from a route value, a query string value, a header, or the request body. But what if you want a handler parameter to be optional? If you have an endpoint like this one,

```
app.MapGet("/stock/{id?}", (int id) => $"Received {id}");
```

given that the route parameter is marked optional, requests to both `/stock/123` and `/stock` will invoke the handler. But in the latter case, there'll be no `id` route value, and you'll get an error like the one shown in figure 7.6.

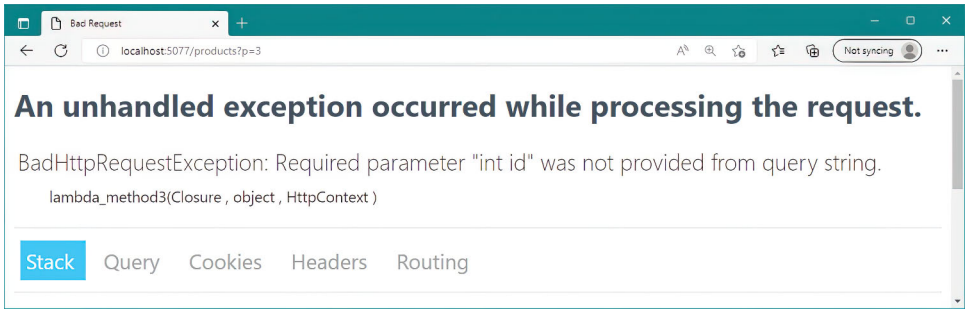


Figure 7.6 If a parameter can't be bound because a value is missing, the `EndpointMiddleware` throws an exception and returns a `400 Bad Request` response. The endpoint handler doesn't run.

The way around this problem is to mark the *handler parameter* as optional by making it nullable. Just as `?` signifies optional in route templates, it signifies optional in the handler parameters. You can update the handler to use `int?` instead of `int`, as shown in the following listing, and the endpoint will handle both `/stock/123` and `/stock` without errors.

Listing 7.5 Using optional parameters in endpoint handlers

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/stock/{id?}", (int? id) => $"Received {id}");

app.MapGet("/stock2", (int? id) => $"Received {id}");

app.MapPost("/stock", (Product? product) => $"Received {product}");

app.Run();
```

Uses a nullable simple type to indicate that the value is optional, so id is null when calling /stock

This example binds to the query string. Id will be null for the request /stock2.

A nullable complex type binds to the body if it's available; otherwise, it's null.

If no corresponding route value or query string contains the required value and the handler parameter is optional, the `EndpointHandler` uses `null` as the argument when invoking the endpoint handler. Similarly, for complex types that bind to the request body, if the request doesn't contain anything in the body and the parameter is optional, the handler will have a `null` argument.

> **WARNING** If the request body contains the literal JSON value `null` and the handler parameter is marked optional, the handler argument will also be `null`. If the parameter isn't marked optional, you get the same error as though the request didn't have a body.

It's worth noting that you mark complex types binding to the request body as optional by using a *nullable reference type* (NRT) annotation: `?`. NRTs, introduced in C# 8, are an attempt to reduce the scourge of null-reference exceptions in C#, colloquially known as "the billion-dollar mistake." See http://mng.bz/vneM.

ASP.NET Core in .NET 7 is built with the assumption that NRTs are enabled for your project (and they're enabled by default in all the templates), so it's worth using them wherever you can. If you choose to disable NRTs explicitly, you may find that some of your types are unexpectedly marked optional, which can lead to some hard-to-debug errors.

> **TIP** Keep NRTs enabled for your minimal API endpoints wherever possible. If you can't use them for your whole project, consider enabling them selectively in Program.cs (or wherever you add your endpoints) by adding `#nullable enable` to the top of the file.

The good news is that ASP.NET Core includes several analyzers built into the compiler to catch configuration problems like the ones described in this section. If you have an optional route parameter but forget to mark the corresponding handler parameter as optional, for example, integrated development environments (IDEs) such as Visual Studio will show a hint, as shown in figure 7.7, and you'll get a build warning. You can read more about the built-in analyzers at http://mng.bz/4DMV.

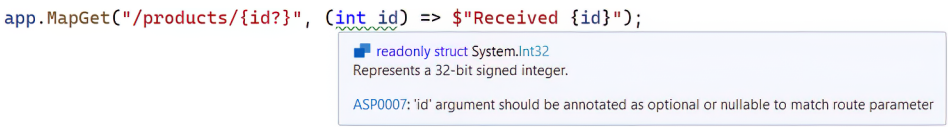


Figure 7.7 Visual Studio and other IDEs use analyzers to detect potential problems with mismatched optionality.

Making your handler parameters optional is one of the approaches you can take, whether they're bound to route parameters, headers, or the query string. Alternatively, you can provide a default value for the parameter as part of the method signature. You can't provide default values for parameters in lambda functions in C# 11,[2] so the following listing shows how to use a local function instead.

Listing 7.6 Using default values for parameters in endpoint handlers

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/stock", StockWithDefaultValue);

app.Run();

string StockWithDefaultValue(int id = 0) => $"Received {id}";
```

The local function StockWithDefaultValue is the endpoint handler.

The id parameter binds to the query string value if it's available; otherwise, it has the value 0.

We've thoroughly covered the differences between simple types and complex types and how they bind. In section 7.6 we look at some special types that don't follow these rules.

[2] C# 12, which will be released with .NET 8, should include support for default values in lambda expressions. For more details, see http://mng.bz/AoRg.

7.6 Binding services and special types

In this section you'll learn how to use some of the special types that you can bind to in your endpoint handlers. By *special*, I mean types that ASP.NET Core is hardcoded to understand or that aren't created from the details of the request, by contrast with the binding you've seen so far. The section looks at three types of parameters:

- Well-known types—that is, hard-coded types that ASP.NET Core knows about, such as `HttpContext` and `HttpRequest`
- `IFormFileCollection` and `IFormFile` for working with file uploads
- Application services registered in `WebApplicationBuilder.Services`

We start by looking at the well-known types you can bind to.

7.6.1 Injecting well-known types

Throughout this book you've seen examples of several well-known types that you can inject into your endpoint handlers, the most notable one being `HttpContext`. The remaining well-known types provide shortcuts for accessing various properties of the `HttpContext` object.

> **NOTE** As described in chapter 3, `HttpContext` acts as a storage box for everything related to a single a request. It contains access to all the low-level details about the request and the response, plus any application services and features you might need.

You can use a well-known type in your endpoint handler by including a parameter of the appropriate type. To access the `HttpContext` in your handler, for example, you could use

```
app.MapGet("/", (HttpContext context) => "Hello world!");
```

You can use the following well-known types in your minimal API endpoint handlers:

- `HttpContext`—This type contains all the details on both the request and the response. You can access everything you need from here, but often, an easier way to access the common properties is to use one of the other well-known types.
- `HttpRequest`—Equivalent to the property `HttpContext.Request`, this type contains all the details about the request only.
- `HttpResponse`—Equivalent to the property `HttpContext.Response`, this type contains all the details about the response only.
- `CancellationToken`—Equivalent to the property `HttpContext.RequestAborted`, this token is canceled if the client aborts the request. It's useful if you need to cancel a long-running task, as described in my post at http://mng.bz/QP2j.

- `ClaimsPrincipal`—Equivalent to the property `HttpContext.User`, this type contains authentication information about the user. You'll learn more about authentication in chapter 23.
- `Stream`—Equivalent to the property `HttpRequest.Body`, this parameter is a reference to the `Stream` object of the request. This parameter can be useful for scenarios in which you need to process large amounts of data from a request efficiently, without holding it all in memory at the same time.
- `PipeReader`—Equivalent to the property `HttpContext.BodyReader`, `PipeReader` provides a higher-level API compared with `Stream`, but it's useful in similar scenarios. You can read more about PipeReader and the System.IO.Pipelines namespace at http://mng.bz/XNY6.

You can access each of the latter well-known types by navigating via an injected `HttpContext` object if you prefer. But injecting the exact object you need generally makes for code that's easier to read.

7.6.2 Injecting services

I've mentioned several times in this book that you need to configure various core services to work with ASP.NET Core. Many services are registered automatically, but often, you must add more to use extra features, such as when you called `AddHttpLogging()` in chapter 3 to add request logging to your pipeline.

> **NOTE** Adding services to your application involves registering them with a dependency injection (DI) container. You'll learn all about DI and registering services in chapters 8 and 9.

You can automatically use any registered service in your endpoint handlers, and ASP.NET Core will inject an instance of the service from the DI container. You saw an example in chapter 6 when you used the `LinkGenerator` service in an endpoint handler. `LinkGenerator` is one of the core services registered by `WebApplicationBuilder`, so it's always available, as shown in the following listing.

Listing 7.7 Using the `LinkGenerator` service in an endpoint handler

```
app.MapGet("/links", (LinkGenerator links) =>
{
    string link = links.GetPathByName("products");
    return $"View the product at {link}";
});
```

The LinkGenerator can be used as a parameter because it's available in the DI container.

Minimal APIs can automatically detect when a service is available in the DI container, but if you want to be explicit, you can also decorate your parameters with the `[FromServices]` attribute:

```
app.MapGet("/links", ([FromServices] LinkGenerator links) =>
```

[FromServices] may be necessary in some rare cases if you're using a custom DI container that doesn't support the APIs used by minimal APIs. But generally, I find that I can keep endpoints readable by avoiding the [From*] attributes wherever possible and relying on minimal APIs to do the right thing automatically.

7.6.3 Binding file uploads with IFormFile and IFormFileCollection

A common feature of many websites is the ability to upload files. This activity could be relatively infrequent, such as a user's uploading a profile picture to their Stack Overflow profile, or it may be integral to the application, such as uploading photos to Facebook.

> **Letting users upload files to your application**
>
> Uploading files to websites is a common activity, but you should consider carefully whether your application *needs* that ability. Whenever users can upload files, the situation is fraught with danger.
>
> You should be careful to treat the incoming files as potentially malicious. Don't trust the filename provided, take care of large files being uploaded, and don't allow the files to be executed on your server.
>
> Files also raise questions about where the data should be stored: in a database, in the filesystem, or in some other storage? None of these questions has a straightforward answer, and you should think hard about the implications of choosing one over the other. Better, don't let users upload files if you don't have to!

ASP.NET Core supports uploading files by exposing the IFormFile interface. You can use this interface in your endpoint handlers, and it will be populated with the details of the file upload:

```
app.MapGet("/upload", (IFormFile file) => {});
```

You can also use an IFormFileCollection if you need to accept multiple files:

```
app.MapGet("/upload", (IFormFileCollection files) =>
{
    foreach (IFormFile file in files)
    {
    }
});
```

The IFormFile object exposes several properties and utility methods for reading the contents of the uploaded file, some of which are shown here:

```
public interface IFormFile
{
    string ContentType { get; }
    long Length { get; }
    string FileName { get; }
    Stream OpenReadStream();
}
```

As you can see, this interface exposes a `FileName` property, which returns the filename that the file was uploaded with. But you know not to trust users, right? You should *never* use the filename directly in your code; users can use it to attack your website and access files that they shouldn't. Always generate a new name for the file before you save it anywhere.

> **WARNING** There are lots of potential threats to consider when accepting file uploads from users. For more information, see http://mng.bz/yQ9q.

The `IFormFile` approach is fine if users are going to be uploading only small files. When your method accepts an `IFormFile` instance, the whole content of the file is buffered in memory and on disk before you receive it. Then you can use the `OpenReadStream` method to read the data out.

If users post large files to your website, you may start to run out of space in memory or on disk as ASP.NET Core buffers each of the files. In that case, you may need to stream the files directly to avoid saving all the data at the same time. Unfortunately, unlike the model-binding approach, streaming large files can be complex and error-prone, so it's outside the scope of this book. For details, see Microsoft's documentation at http://mng.bz/MBgn.

> **TIP** Don't use the `IFormFile` interface to handle large file uploads, as you may see performance problem. Be aware that you can't rely on users *not* to upload large files, so avoid file uploads when you can!

For the vast majority of minimal API endpoints, the default configuration of model binding for simple and complex types works perfectly well. But you may find some situations in which you need to take a bit more control.

7.7 Custom binding with BindAsync

The model binding you get out of the box with minimal APIs covers most of the common situations that you'll run into when building HTTP APIs, but there are always a few edge cases in which you can't use it.

You've already seen that you can inject `HttpContext` into your endpoint handlers, so you have direct access to the request details in your handler, but often, you still want to encapsulate the logic for extracting the data you need. You can get the best of both worlds in minimal APIs by implementing `BindAsync` in your endpoint handler parameter types and taking advantage of completely custom model binding. To add custom binding for a parameter type, you must implement one of the following two `static BindAsync` methods in your type `T`:

```
public static ValueTask<T?> BindAsync(HttpContext context);
public static ValueTask<T?> BindAsync(
HttpContext context, ParameterInfo parameter);
```

Both methods accept an `HttpContext`, so you can extract anything you need from the request. But the latter case also provides reflection details about the parameter you're binding. In most cases the simpler signature should be sufficient, but you never know!

Listing 7.8 shows an example of using `BindAsync` to bind a record to the request body by using a custom format. The implementation shown in the listing assumes that the body contains two `double` values, with a line break between them, and if so, it successfully parses the `SizeDetails` object. If there are any problems along the way, it returns `null`.

Listing 7.8 Using `BindAsync` for custom model binding

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/sizes", (SizeDetails size) => $"Received {size}");

app.Run();

public record SizeDetails(double height, double width)
{
    public static async ValueTask<SizeDetails?> BindAsync(
        HttpContext context)
    {
        using var sr = new StreamReader(context.Request.Body);

        string? line1 = await sr.ReadLineAsync(context.RequestAborted);
        if (line1 is null) { return null; }

        string? line2 = await sr.ReadLineAsync(context.RequestAborted);
        if (line2 is null) { return null; }

        return double.TryParse(line1, out double height)
            && double.TryParse(line2, out double width)
            ? new SizeDetails(height, width)
            : null;
    }
}
```

No extra attributes are needed for the SizeDetails parameter, as it has a BindAsync method.

SizeDetails implements the static BindAsync method.

Creates a StreamReader to read the request body

Reads a line of text from the body

If either line is null, indicating no content, stops processing

Tries to parse the two lines as doubles

If the parsing is successful, creates the SizeDetails model and returns it . . .

. . . otherwise, returns null

In listing 7.8 we return `null` if parsing fails. The endpoint shown will cause the `EndpointMiddleware` to throw a `BadHttpRequestException` and return a `400` error, because the size parameter in the endpoint is required (not marked optional). You could have thrown an exception in `BindAsync`, but it wouldn't have been caught by the `EndpointMiddleware` and would have resulted in a `500` response.

7.8 Choosing a binding source

Phew! We've finally covered all the ways you can bind a request to parameters in minimal APIs. In many cases, things should work as you expect. Simple types such as `int` and `string` bind to route values and query string values by default, and complex types

bind to the request body. But it can get confusing when you add attributes, `BindAsync`, and `TryParse` to the mix!

When the minimal API infrastructure tries to bind a parameter, it checks all the following binding sources in order. The first binding source that matches is the one it uses:

1. If the parameter defines an explicit binding source using attributes such as `[FromRoute]`, `[FromQuery]`, or `[FromBody]`, the parameter binds to that part of the request.
2. If the parameter is a well-known type such as `HttpContext`, `HttpRequest`, `Stream`, or `IFormFile`, the parameter is bound to the corresponding value.
3. If the parameter type has a `BindAsync()` method, use that method for binding.
4. If the parameter is a `string` or has an appropriate `TryParse()` method (so is a simple type):
   a. If the name of the parameter matches a route parameter name, bind to the route value.
   b. Otherwise, bind to the query string.
5. If the parameter is an array of simple types, a `string[]`, or `StringValues`, and the request is a `GET` or similar HTTP verb that normally doesn't have a request body, bind to the query string.
6. If the parameter is a known service type from the dependency injection container, bind by injecting the service from the container.
7. Finally, bind to the body by deserializing from JSON.

The minimal API infrastructure follows this sequence for every parameter in a handler and stops at the first matching binding source.

> **WARNING** If binding fails for the entry, and the parameter isn't optional, the request fails with a `400 Bad Request` response. The minimal API doesn't try another binding source after one source fails.

Remembering this sequence of binding sources is one of the hardest things about minimal APIs to get your head around. If you're struggling to work out why a request isn't working as you expect, be sure to come back and check this sequence. I once had a parameter that wasn't binding to a route parameter, despite its having a `TryParse` method. When I checked the sequence, I realized that it also had a `BindAsync` method that was taking precedence!

7.9 Simplifying handlers with AsParameters

Before we move on, we'll take a quick look at a .NET 7 feature for minimal APIs that can simplify some endpoint handlers: the `[AsParameters]` attribute. Consider the following `GET` endpoint, which binds to a route value, a header value, and some query values:

```
app.MapGet("/category/{id}", (int id, int page, [FromHeader(Name = "sort")]
➥ bool? sortAsc, [FromQuery(Name = "q")] string search) => { });
```

I think you'll agree that the handler parameters for this method are somewhat hard to read. The parameters define the expected shape of the request, which isn't ideal. The `[AsParameters]` attribute lets you wrap all these arguments into a single `class` or `struct`, simplifying the method signature and making everything more readable.

Listing 7.9 shows an example of converting this endpoint to use `[AsParameters]` by replacing it with a `record struct`. You could also use a `class`, `record`, or `struct`, and you can use properties instead of constructor parameters if you prefer. See the documentation for all the permutations available at http://mng.bz/a1KB.

Listing 7.9 Using `[AsParameters]` to simplify endpoint handler parameters

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapGet("/category/{id}",
    ([AsParameters] SearchModel model) => $"Received {model}");

app.Run();

record struct SearchModel(
    int id,
    int page,
    [FromHeader(Name = "sort")] bool? sortAsc,
    [FromQuery(Name = "q")] string search);
```

[AsParameters] indicates that the constructor or properties of the type should be bound, not the type itself.

Each parameter is bound as though it were written in the endpoint handler.

The same attributes and rules apply for binding an `[AsParameters]` type's constructor parameters and binding endpoint handler parameters, so you can use `[From*]` attributes, inject services and well-known types, and read from the body. This approach can make your endpoints more readable if you find that they're getting a bit unwieldy.

> **TIP** In chapter 16 you'll learn about model binding in MVC and Razor Pages. You'll be pleased to know that in those cases, the `[AsParameters]` approach works out of the box without the need for an extra attribute.

That brings us to the end of this section on model binding. If all went well, your endpoint handler's arguments are created, and the handler is ready to execute its logic. It's time to handle the request, right? Nothing to worry about.

Not so fast! How do you know that the data you received was valid? How do you know that you haven't been sent malicious data attempting a SQL injection attack or a phone number full of letters? The binder is relatively blindly assigning values sent in a request, which you're happily going to plug into your own methods. What stops nefarious little Jimmy from sending malicious values to your application? Except for basic safeguards, nothing is stopping him, which is why it's important that you *always* validate the input coming in. ASP.NET Core provides a way to do this in a declarative manner out of the box, which is the focus of section 7.10.

7.10 Handling user input with model validation

In this section, I discuss the following topics:

- What validation is and why you need it
- How to use `DataAnnotations` attributes to describe the data you expect
- How to validate your endpoint handler parameters

Validation in general is a big topic, one that you'll need to consider in every app you build. Minimal APIs don't include validation by default, instead opting to provide nonprescriptive hooks via the filters you learned about in chapter 5. This design gives you multiple options for adding validation to your app; be sure that you do add some!

7.10.1 The need for validation

Data can come from many sources in your web application. You could load data from files, read it from a database, or accept values that are sent in a request. Although you may be inclined to trust that the data already on your server is valid (though this assumption is sometimes dangerous!), you *definitely* shouldn't trust the data sent as part of a request.

> **TIP** You can read more about the goals of validation, implementation approaches, and potential attacks at http://mng.bz/gBxE.

You should validate your endpoint handler parameters before you use them to do anything that touches your domain, anything that touches your infrastructure, or anything that could leak information to an attacker. Note that this warning is intentionally vague, as there's no defined point in minimal APIs where validation should occur. I advise that you do it as soon as possible in the minimal API filter pipeline.

Always validate data provided by users before you use it in your methods. You have no idea what the browser may have sent you. The classic example of little Bobby Tables (https://xkcd.com/327) highlights the need to always validate data sent by a user.

Validation isn't used only to check for security threats, though. It's also needed to check for nonmalicious errors:

- *Data should be formatted correctly.* Email fields have a valid email format, for example.
- *Numbers may need to be in a particular range.* You can't buy -1 copies of this book!
- *Some values may be required, but others are optional.* Name may be required for a profile, but phone number is optional.
- *Values must conform to your business requirements.* You can't convert a currency to itself; it needs to be converted to a different currency.

As mentioned earlier, the minimal API framework doesn't include anything specific to help you with these requirements, but you can use filters to implement validation, as you'll see in section 7.10.3. .NET 7 also includes a set of attributes that you can use to simplify your validation code significantly.

7.10.2 Using DataAnnotations attributes for validation

Validation attributes—more precisely, `DataAnnotations` attributes—allow you to specify the rules that your parameters should conform to. They provide metadata about a parameter type by describing the *sort* of data the binding model should contain, as opposed to the data itself.

You can apply `DataAnnotations` attributes directly to your parameter types to indicate the type of data that's acceptable. This approach allows you to check that required fields have been provided, that numbers are in the correct range, and that email fields are valid email addresses, for example.

Consider the checkout page for a currency-converter application. You need to collect details about the user—their name, email, and (optionally) phone number—so you create an API to capture these details. The following listing shows the outline of that API, which takes a `UserModel` parameter. The `UserModel` type is decorated with validation attributes that represent the validation rules for the model.

Listing 7.10 Adding `DataAnnotations` to a type to provide metadata

```
using System.ComponentModel.DataAnnotations;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/users", (UserModel user) => user.ToString());

app.Run();

public record UserModel
{
    [Required]
    [StringLength(100)]
    [Display(Name = "Your name")]
    public string FirstName { get; set; }

    [Required]
    [StringLength(100)]
    [Display(Name = "Last name")]
    public string LastName { get; set; }

    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Phone]
    [Display(Name = "Phone number")]
    public string PhoneNumber { get; set; }
}
```

Adds this using statement to use the validation attributes

The API takes a UserModel parameter and binds it to the request body.

Values marked Required must be provided.

The StringLengthAttribute sets the maximum length for the property.

Customizes the name used to describe the property

Validates that the value of Email may be a valid email address

Validates that the value of PhoneNumber has a valid telephone number format

Suddenly, your parameter type, which was sparse on details, contains a wealth of information. You've specified that the `FirstName` property should always be provided; that it

should have a maximum length of 100 characters; and that when it's referred to (in error messages, for example), it should be called `"Your name"` instead of `"FirstName"`.

The great thing about these attributes is that they clearly declare the *expected* state of an instance of the type. By looking at these attributes, you know what the properties will contain, or at least *should* contain. Then you can then write code after model binding to confirm that the bound parameter is valid, as you'll see in section 7.10.3.

You've got a plethora of attributes to choose among when you apply `DataAnnotations` to your types. I've listed some of the common ones here, but you can find more in the `System.ComponentModel.DataAnnotations` namespace. For a more complete list, I recommend using IntelliSense in your IDE or checking the documentation at http://mng.bz/e1Mv.

- `[CreditCard]`—Validates that a property has a valid credit card format
- `[EmailAddress]`—Validates that a property has a valid email address format
- `[StringLength(max)]`—Validates that a string has at most `max` number of characters
- `[MinLength(min)]`—Validates that a collection has at least the `min` number of items
- `[Phone]`—Validates that a property has a valid phone number format
- `[Range(min, max)]`—Validates that a property has a value between `min` and `max`
- `[RegularExpression(regex)]`—Validates that a property conforms to the `regex` regular expression pattern
- `[Url]`—Validates that a property has a valid URL format
- `[Required]`—Indicates that the property must not be `null`
- `[Compare]`—Allows you to confirm that two properties have the same value (such as `Email` and `ConfirmEmail`)

WARNING The `[EmailAddress]` and `[Phone]` attributes validate only that the *format* of the value is potentially correct. They don't validate that the email address or phone number exists. For an example of how to do more rigorous phone number validation, see this post on the Twilio blog: http://mng.bz/xmZe.

The `DataAnnotations` attributes aren't new; they've been part of the .NET Framework since version 3.5, and their use in ASP.NET Core is almost the same as in the previous version of ASP.NET. They're also used for purposes other than validation. Entity Framework Core (among others) uses `DataAnnotations` to define the types of columns and rules to use when creating database tables from C# classes. You can read more about Entity Framework Core in chapter 12 and in *Entity Framework Core in Action*, 2nd ed., by Jon P. Smith (Manning, 2021).

If the `DataAnnotation` attributes provided out of the box don't cover everything you need, it's possible to write custom attributes by deriving from the base `ValidationAttribute`. You'll see how to create a custom validation attribute in chapter 32.

One common limitation with `DataAnnotation` attributes is that it's hard to validate properties that depend on the values of other properties. Maybe the `UserModel` type

from listing 7.10 requires you to provide either an email address or a phone number but not both, which is hard to achieve with attributes. In this type of situation, you can implement `IValidatableObject` in your models instead of, or in addition to, using attributes. In listing 7.11, a validation rule is added to `UserModel` whether the email or phone number is provided. If it isn't, `Validate()` returns a `ValidationResult` describing the problem.

Listing 7.11 Implementing `IValidatableObject`

```
using System.ComponentModel.DataAnnotations;

public record CreateUserModel : IValidatableObject
{
    [EmailAddress]
    public string Email { get; set; }

    [Phone]
    public string PhoneNumber { get; set; }

    public IEnumerable<ValidationResult> Validate(
        ValidationContext validationContext)
    {
        if(string.IsNullOrEmpty(Email)
            && string.IsNullOrEmpty(PhoneNumber))
        {
            yield return new ValidationResult(
                "You must provide an Email or a PhoneNumber",
                New[] { nameof(Email), nameof(PhoneNumber) });
        }
    }
}
```

Implements the IValidatableObject interface

The DataAnnotation attributes continue to validate basic format requirements.

Validate is the only function to implement in IValidatableObject.

Checks whether the object is valid . . .

. . . and if not, returns a result describing the error

`IValidatableObject` helps cover some of the cases that attributes alone can't handle, but it's not always the best option. The `Validate` function doesn't give easy access to your app's services, and the function executes only if all the `DataAnnotation` attribute conditions are met.

> **TIP** `DataAnnotations` are good for input validation of properties in isolation but not so good for validating complex business rules. You'll most likely need to perform this validation outside the `DataAnnotations` framework.

Alternatively, if you're not a fan of the `DataAnnotation` attribute-based-plus-`IValidatableObject` approach, you could use the popular FluentValidation library (https://github.com/JeremySkinner/FluentValidation) in your minimal APIs instead. Minimal APIs are completely flexible, so you can use whichever approach you prefer.

`DataAnnotations` attributes provide the basic metadata for validation, but no part of listing 7.10 or listing 7.11 uses the validation attributes you added. You still need to add code to read the parameter type's metadata, check whether the data is valid, and return an error response if it's invalid. ASP.NET Core doesn't include a dedicated

validation API for that task in minimal APIs, but you can easily add it with a small NuGet package.

7.10.3 Adding a validation filter to your minimal APIs

Microsoft decided not to include any dedicated validation APIs in minimal APIs. By contrast, validation is a built-in core feature of Razor Pages and MVC. Microsoft's reasoning was that the company wanted to provide flexibility and choice for users to add validation in the way that works best for them, but didn't want to affect performance for those who didn't want to use their implementation.

Consequently, validation in minimal APIs typically relies on the filter pipeline. As a classic cross-cutting concern, validation is a good fit for a filter. The only downside is that typically, you need to write your own filter rather than use an existing API. The positive side is that validation gives you complete flexibility, including the ability to use an alternative validation library (such as FluentValidation) if you prefer.

Luckily, Damian Edwards, a project manager architect on the ASP.NET Core team at Microsoft, has a NuGet package called MinimalApis.Extensions that provides the filter for you. Using a simple validation system that hooks into the `DataAnnotations` on your models, this NuGet package provides an extension method called `WithParameterValidation()` that you can add to your endpoints. To add the package, search for MinimalApis.Extensions from the NuGet Package Manager in your IDE (be sure to include prerelease versions), or run the following, using the .NET command-line interface:

```
dotnet add package MinimalApis.Extensions
```

After you've added the package, you can add validation to any of your endpoints by adding a filter using `WithParameterValidation()`, as shown in listing 7.12. After the `UserModel` is bound to the JSON body of the request, the validation filter executes as part of the filter pipeline. If the `user` parameter is valid, execution passes to the endpoint handler. If the parameter is invalid, a `400 Bad Request` Problem Details response is returned containing a description of the errors, as shown in figure 7.8.

Listing 7.12 Adding validation to minimal APIs using MinimalApis.Extensions

```
using System.ComponentModel.DataAnnotations;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/users", (UserModel user) => user.ToString())
    .WithParameterValidation();

app.Run();

public record UserModel
{
    [Required]
```

Adds the validation filter to the endpoint

The UserModel defines its validation requirements using DataAnnotations attributes.

```
    [StringLength(100)]
    [Display(Name = "Your name")]
    public string Name { get; set; }

    [Required]
    [EmailAddress]
    public string Email { get; set; }
}
```

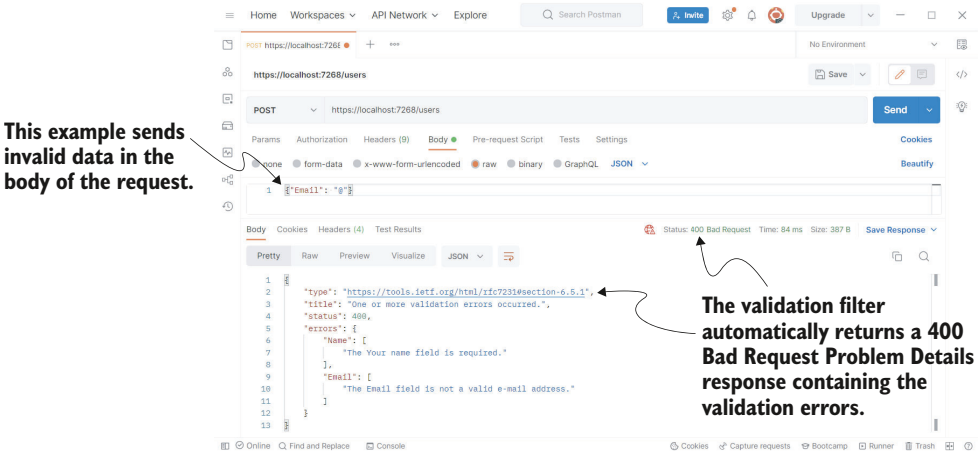


Figure 7.8 If the data sent in the request body is not valid, the validation filter automatically returns a `400 Bad Request` response, containing the validation errors, and the endpoint handler doesn't execute.

Listing 7.12 shows how you can validate a complex type, but in some cases, you may want to validate simple types. You may want to validate that the `id` value in the following handler should be between `1` and `100`:

```
app.MapGet("/user/{id}", (int id) => $"Received {id}")
    .WithParameterValidation();
```

Unfortunately, that's not easy to do with `DataAnnotations` attributes. The validation filter will check the `int` type, see that it's not a type that has any `DataAnnotations` on its properties, and won't validate it.

> **WARNING** Adding attributes to the handler, as in `([Range(1, 100)] int id)`, doesn't work. The attributes here are added to the *parameter*, not to properties of the `int` type, so the validator won't find them.

There are several ways around this problem, but the simplest is to use the `[AsParameters]` attribute you saw in section 7.9 and apply annotations to the model. The following listing shows how.

Listing 7.13 Adding validation to minimal APIs using MinimalApis.Extensions

```
using System.ComponentModel.DataAnnotations;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();

app.MapPost("/user/{id}",
    ([AsParameters] GetUserModel model) => $"Received {model.Id}")
   .WithParameterValidation();

app.Run();

struct GetUserModel
{
    [Range(1, 10)]
    Public int Id { get; set; }
}
```

Uses [AsParameters] to create a type than can be validated

Adds the validation filter to the endpoint

Adds validation attributes to your simple types

That concludes this look at model binding in minimal APIs. You saw how the ASP.NET Core framework uses model binding to simplify the process of extracting values from a request and turning them into normal .NET objects you can work with quickly. The many ways to bind may be making your head spin, but normally, you can stick to the basics and fall back to the more complex types as and when you need them.

Although the discussion is short, the most important aspect of this chapter is its focus on validation—a common concern for all web applications. Whether you choose to use `DataAnnotations` or a different validation approach, you must make sure to validate any data you receive in all your endpoints.

In chapter 8 we leave minimal APIs behind to look at dependency injection in ASP.NET Core and see how it helps create loosely coupled applications. You'll learn how to register the ASP.NET Core framework services with a container, add your own services, and manage service lifetimes.

Summary

- Model binding is the process of creating the arguments for endpoint handlers from the details of an HTTP request. Model binding takes care of extracting and parsing the strings in the request so that you don't have to.
- Simple values such as `int`, `string`, and `double` can bind to route values, query string values, and headers. These values are common and easy to extract from the request without any manual parsing.
- If a simple value fails to bind because the value in the request is incompatible with the handler parameter, a `BadHttpRequestException` is thrown, and a `400 Bad Request` response is returned.
- You can turn a custom type into a simple type by adding a `TryParse` method with the signature `bool TryParse(string value, out T result)`. If you return `false` from this method, minimal APIs will return a `400 Bad Request` response.

- Complex types bind to the request body by default by deserializing from JSON. Minimal APIs can bind only to JSON bodies; you can't use model binding to access form values.
- By default, you can't bind the body of `GET` requests, as that goes against the expectations for `GET` requests. Doing so will cause an exception at runtime.
- Arrays of simple types bind by default to query string values for `GET` requests and to the request body for `POST` requests. This difference can cause confusion, so always consider whether an array is the best option.
- All the parameters of a handler must bind correctly. If a parameter tries to bind to a missing value, you'll get a `BadHttpRequestException` and a `400 Bad Request` response.
- You can use well-known types such as `HttpContext` and any services from the dependency injection container in your endpoint handlers. Minimal APIs check whether each complex type in your handler is registered as a service in the DI container; if not, they treat it as a complex type to bind to the request body instead.
- You can read files sent in the request by using the `IFormFile` and `IFormFileCollection` interfaces in your endpoint handlers. Take care accepting file uploads with these interfaces, as they can open your application to attacks from users.
- You can completely customize how a type binds by using custom binding. Create a static function with the signature `public static ValueTask<T?> BindAsync(HttpContext context)`, and return the bound property. This approach can be useful for handling complex scenarios, such as arbitrary JSON uploads.
- You can override the default binding source for a parameter by applying `[From*]` attributes to your handler parameters, such as `[FromHeader]`, `[FromQuery]`, `[FromBody]`, and `[FromServices]`. These parameters take precedence over convention-based assumptions.
- You can encapsulate an endpoint handler's parameters by creating a type containing all the parameters as properties or a constructor argument and decorate the parameter with the `[AsParameters]` attribute. This approach can help you simplify your endpoint's method signature.
- Validation is necessary to check for security threats. Check that data is formatted correctly, confirm that it conforms to expected values and verify that it meets your business rules.
- Minimal APIs don't have built-in validation APIs, so you typically apply validation via a minimal API filter. This approach provides flexibility, as you can implement validation in the way that suits you best, though it typically means that you need to use a third-party package.
- The MinimalApis.Extensions NuGet package provides a validation filter that uses `DataAnnotations` attributes to declaratively define the expected values. You can add the filter with the extension method `WithParameterValidation()`.
- To add custom validation of simple types with MinimalApis.Extensions, you must create a containing type and use the `[AsParameters]` attribute.